European Journal of Operational Research 240 (2015) 32–42

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

Heuristics and lower bounds for the simple assembly line balancing problem type 1: Overview, computational tests and improvements

Tom Pape *

Clinical Operational Research Unit, University College London, 4 Taviton Street, WC1H 0BT London, UK

ARTICLE INFO

Article history:
Received 24 November 2012
Accepted 19 June 2014
Available online 1 July 2014

Keywords:
Heuristic
Lower bound
Assembly line balancing
Reduction technique
Partitioning problem

ABSTRACT

Assigning tasks to work stations is an essential problem which needs to be addressed in an assembly line design. The most basic model is called simple assembly line balancing problem type 1 (SALBP-1). We provide a survey on 12 heuristics and 9 lower bounds for this model and test them on a traditional and a lately-published benchmark dataset. The present paper focuses on algorithms published before 2011.

We improve an already existing dynamic programming and a tabu search approach significantly. These two are also identified as the most effective heuristics; each with advantages for certain problem characteristics. Additionally we show that lower bounds for SALBP-1 can be distinctly sharpened when merging them and applying problem reduction techniques.

1. Introduction

Assembly lines are a common way to organise mass production of standardised products. They consist of ordered stations along a conveyor belt to which a set of tasks is assigned to. The cycle time determines how much time the stations' workers and/or machines have to fulfil their tasks before passing on the workpiece to the next following station.

The simple assembly line balancing problem type 1 (SALBP-1) is a fundamental and well-studied problem of assembly line design (Baybars, 1986; Scholl, 1999). The tasks $j = 1,\ldots,n$ are defined by task times t_j and their positions within the precedence graph. The goal is to minimise m as number of loaded stations given a fixed cycle time c. A list of all used symbols can be found in Table 1. Fig. 1 illustrates SALBP-1 exemplarily. The nodes (tasks) of the precedence graph are indexed from 1 to 8 and above them stand their task times t_j. For SALBP-1 a solution is feasible if (i) the tasks of each station do not have a task time sum larger than c and (ii) no direct or indirect predecessor of any task j is assigned to a later station than j is assigned to. The shaded regions identify a possible feasible solution with 4 stations. If one turns around the arrows' directions in the precedence graph, one receives the reverse problem. A solution of the reverse problem (backward direction) is always a feasible solution of the original SALBP-1 (forward direction) after turning around the station order.

Many general assembly line balancing problems (GALBPs) base on this simple logic and extend it, for example, with ergonomic considerations, space restraints and mixed-model production. Therefore algorithms should be analysed properly on their effectiveness on SALBP-1 before adapting them to more sophisticated GALBPs. Comparing the effectiveness of procedures only with the results reported in their original papers may be distorting due to different computational environments, incomparable CPU times, or different datasets. This explains the need for thoroughly conducted comparing studies. By now those exist only for some exact procedures (Baybars, 1986; Scholl & Klein, 1999), simple algorithms (Ponnambalam, Aravindan, & Mogileeswar Naidu, 1999) and priority rules (Otto, Otto, & Scholl, 2014; Scholl & Voß, 1996).

SALBP-1 is NP-hard (Karp, 1972), so that heuristics are essential to obtain upper bounds for problems. Furthermore in order to assess the quality of found solutions, lower bounds methods are important in integer optimisation. Closing the research gap by a comparing study of upper and lower bounds for SALBP-1 is the first and main goal of this paper.

The second goal is the improvement of some already-known procedures, namely tabu search, dynamic programming, lower bound 7 and 8, as well as SALBP-1 reduction techniques. It will also be discussed how to use problem reduction techniques for sharpening lower bounds.

As benchmark dataset this study uses the collection of 269 instances from Scholl (1993) as well as the new systematically-generated 100-tasks and 1000-tasks problems from Otto, Otto, and Scholl (2013) denoted as SCHOLL, OTTO-100 and OTTO-1000, respectively, in the following. By now, there has not been thor-

* Tel.: +44 7440153248.
E-mail address: t.pape@ucl.ac.uk

http://dx.doi.org/10.1016/j.ejor.2014.06.023
0377-2217/

Table 1
Symbols for SALBP-1.

c	Cycle times
j	Index of the tasks
$J(a, b]$	Set of all tasks with $a < p_j \leqslant b$
k	Index of the stations
m	Number of stations
n	Number of tasks
p_j	t_j/c_j
$\left(P_j^*\right)P_j$	(Direct) predecessors of j
$\left(S_j^*\right)S_j$	(Direct) successors of j
S_k	Load of station k
t_j	Task time of j

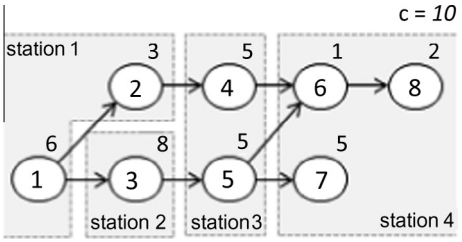


Fig. 1. SALBP-1 with a grey-shaded solution.

oughly investigated in how far the success of SALBP-1 algorithms depends on the problem properties. Finding an answer to this question is the paper's third goal.

The article is organised as follows: Section 2 outlines the "General idea" of each examined heuristic briefly, states some "Experience" the author made during implementation and describes methodical improvements for some approaches. The "General ideas" require some knowledge about standard solution procedures in operational research and the "Experience" can usually not be fully understood without having read the original papers before. Section 3 explains the improvements proposed for lower bounds. Section 4 reports the computational result. Section 5 summarises and discusses the main findings.

2. Heuristics

2.1. Falkenauer and Delchambre (1992): GA-FD

General idea: Falkenauer and Delchambre designed a genetic algorithm (GA) in which the genes are the station loads of the solution (chromosome). Thereby the genes on the chromosome are not necessarily ordered in the sequence of the stations in the final solution. Instead the precedence relations between the genes are kept in an additional genes' precedence graph. This encoding technique is called group encoding. Falkenauer and Delchambre apply some of the usual genetic operators on the genes to obtain improved children. Thereby they use a fitness function which rewards well-filled stations more than it punishes less filled ones in a solution. After crossover the children become usually infeasible since tasks are assigned to more than one station and the precedence relations are violated. A complex healing process must follow therefore.

Experience: Falkenauer and Delchambre's proposed strategy to make children feasible (eliminating cycles in the genes' precedence graph) needed often more than 500,000 iterations (>1 minute CPU time) on OTTO-100 just to obtain one new solution. In our experiments, the time limit was often reached before repairing the children of the first crossover.

2.2. Sabuncuoglu, Erel, and Tanyer (2000): GA-S

General idea: Sabuncuoglu et al. introduce a genetic algorithm in which the tasks as genes are always ordered on the chromosome in a sequence that obeys the precedence graph (order encoding). They apply a crossover technique which completely avoids infeasibility. In contrast to GA-FD, chromosomes with equally loaded stations receive the highest fitness score. Additionally they apply a freezing technique which finalises the task assignments of the current first and last unfrozen station if they provide satisfying loads. It shall be noted that a better performing hybrid genetic algorithm incorporating priority rules and local search is published in Gonçalves and De Almeida (2002).

Experience: Due to the smart crossover technique, GA-S is fast in creating new solutions (about 5000 per second on OTTO-100). The freezing often leads to a search stop before reaching an optimal solution or the given time limit. In those cases we revoke the algorithm with a twice that strict freezing policy. After preliminary tests we chose a population size of 100, a mutation probability of 1%, a final replacing probability of 1%, and an initial freezing parameter DPC of 10% which halves itself every time all stations are frozen.

2.3. Nearchou (2005): DEA

General idea: Nearchou designed a differential evolutionary algorithm (DEA) to tackle SALBP-1. We assume that learning about the tasks' variability concerning their positions in good solutions is its main idea. The solutions (chromosomes) are sub-range encoded what can be linearly transformed into order encoding. Sub-ranges express an order position of a task by small float-point intervals between 0 and 1. For instances the solution $(0.4, 0.32, 0.7)$ basing on the sub ranges $[0, \frac{1}{3}) \rightarrow 1$, $[\frac{1}{3}, \frac{2}{3}) \rightarrow 2$ and $[\frac{2}{3}, 1) \rightarrow 3$ would mean an order encoding $(2, 1, 3)$. Crossover works like in genetic algorithms, just mutation – which is performed in every iteration – is done differently. Given three sub-range encoded solutions x_a, x_b, x_c of the population as vectors, one receives the mutant with $x_m = x_c + w(x_a - x_b)$ where w denotes a small weight. Solutions after mutation and crossover need to be healed of infeasibility.

Experience: Repairing infeasible solutions makes up 85% of the CPU time and leads to a low number of created solutions per second (about 60 per second on OTTO-100). For the parameters w, crossover probability and population size we opted for 0.3, 1 and 100, respectively, after some tests. Nearchou demands a replacement of some solutions when the population becomes too homogeneous but does not define how he measures homogeneity. Therefore, we replace one solution from the bottom half of the population space according to their fitness with a new random one after every five iterations.

2.4. Bautista and Pereira (2002): ACO

Bautista and Pereira test several versions of ant colony optimisation (ACO) for SALBP-1. Here only Bautista and Pereira's best version (task-position policy, summed trail reading) is described and implemented. Very similar designs can be found in Boysen and Fliedner (2008) and Zhang, Cheng, Tang, and Zhong (2007). An ant colony algorithm which strongly relies on a local search is published in Bautista and Pereira (2007).

General idea: Solutions are order encoded, and between each task j and each order position o exists a pheromone trail τ_{jo}. Solutions are constructed either in forward or in backward direction by adding one task after another. To select the next task for order position $\bar{o}$ from all those without unassigned predecessors, a roulette wheel selection based on the task preferences is conducted. The task preference is the weighted product of the pheromone trails from order position 1 to $\bar{o}$, i.e. $\sum_{o=1}^{\bar{o}} \tau_{jo}$, and a normalised task

priority. As fitness function for updating τ_{jo}, Bautista and Pereira simply opt for the number of stations.

Experience: To mention is their coarse-grained fitness function what Bautista and Pereira themselves admit as not perfect. Therefore the fitness function from Zhang et al. (2007) which rewards, like in GA-FD, well-loaded stations more than it punishes little-loaded ones is used here. Due to this change, 187 instead of 181 optimal solutions could be detected after 30 seconds on SCHOLL. It shall also be mentioned that their steady normalisation of the task priorities counts for one-quarter of the CPU time but could be easily replaced by Scholl (1999) and Baybars (1986)-bandwidth scaled task priorities (not done in this study). Furthermore, restarting ACO 12 times with 15 seconds CPU time on OTTO-100 provides a best solution with on average 0.08 stations less than executing ACO once with 180 seconds CPU time on OTTO-100. This observation provides some evidence that the current ACO design does not explore the whole solution space sufficiently.

2.5. Scholl and Voß (1996): Tabu-SV

General idea: As main feature, the algorithm attempts to find feasible SALBP-1 solutions by shifting and swapping tasks in a way which eliminates the exceeding of the cycle time ("overload"). The overload o_k for station k is defined as $\max\{0, t(S_k) - c\}$. That means solutions which exceed the cycle time in some stations are allowed but not those which violate the precedence relations. The last station to which any task of P_j^* is assigned to and the earliest station to which any task of S_j^* is assigned to set the boundaries within which task j can be shifted to. Two tasks can be swapped if each is within the shifting boundaries of the other one. Now to the steps of Tabu-SV: The procedure is initialised with a feasible solution of the related SALBP-2 (minimising the cycle time c given m as number of stations) with m equal to a lower bound of SALBP-1. The SALBP-2 solution is found with a simple priority rule. Afterwards, each iteration consists of 4 steps. (1) A highly overloaded station k^* with $o_{k^*} = \max_k\{o_k\}$ is randomly selected as "base station". (2) For each task j in this base station, all possible shifts to and swaps with other stations are evaluated with respect to lowering the maximum station overload $\max_k\{o_k\}$. (3) The best move is gone and (4) the old assignment of the moved task(s) is set tabu for a certain dynamically adapted number of iterations. If through shifts and swaps no feasible SALBP-1 solution is detected after a certain number of iterations, m is increased by one. When a feasible SALBP-1 solution is found for the first time, m is decreased by one and never raised again. Additionally Scholl and Voß apply some advanced tabu search policies. Their version with EUREKA is not tested here.

Experience: Our implementation – which adjusts parameters linearly depending on the time and not the iterations – detected a cycling in the Tabu-SV search for several instances. It is caused by the quasi-static selection of the base station. Therefore, Tabu-SV yielded – as similarly reported by Scholl and Voß – maximal 200 optimal solutions on SCHOLL within a few seconds but never comes further.

2.6. Simple-Tabu

To avoid cycling in Tabu-SV, two changes are proposed. First, the base station is now selected directly proportional to the quadratic station time overloads o_k^2 via roulette wheel selection in 90% of the cases. Because this alone cannot avoid cycling completely, the base station is selected randomly from all stations $k = 1 \ldots m$ in the remaining cases. Second, solutions are now assessed according to the fitness function $\sum_{k=1}^{m} o_k^2$ which measures the sum of the quadratic overloads over all stations. Through the changes Scholl and Voß' conflict management, moving gap strategy and adaptive tabu length lose their influences on the results and may be omitted. For 1000-tasks instances and CPU times under 30 seconds, their initial simple priority rule often provides better results than executing Tabu-SV with a random initial solution. Since this survey does not want to compare priority rules, it is started with a random solution instead. So, one ends up with a Simple-Tabu search: Beginning with a randomly generated feasible solution with m stations, the lowest-loaded station is selected, completely emptied by shifts and afterwards closed. One receives a solution with $m - 1$ stations which obeys the precedence graph but usually not the cycle time. In each iteration a base station is selected as described above, all possible shifts and swaps for it examined in the same way as in Scholl and Voß and the best one according to the new fitness function gone. If a task j is moved from station k to station k', task j is not allowed to move back to k for 10 iterations, i.e. the task-station combination (j, k) is tabu for the next 10 iterations. This tabu rule is only disabled if a shift or a swap brings a fitness value lower than ever reached for $m - 1$ stations before (aspiration criterion from Scholl & Voß (1996)). When a solution with fitness value of zero is found, it is feasible for SALBP-1. Now again the lowest-loaded station is emptied and closed and the procedure continues.

2.7. Lapierre, Ruiz, and Soriano (2006): Tabu-L

General idea: From Tabu-SV Lapierre et al. adopted the idea of allowing overloads but no precedence violations. Tabu-L is initialised with a randomly generated feasible SALBP-1 solution. In each iteration, it is decided at first whether to test either shifts or swaps, then a suitable station is selected, and finally the best shift (or swap) according to their fitness function is gone. The shift mainly selects little-loaded stations and tries to empty them by assigning one of their tasks to another station. The swap chooses approximately half-loaded stations and attempts to fill them by exchanging one of their tasks with a task of another station. The fitness of a solution is measured by a function which (i) rewards well-loaded stations more than it punishes little-loaded ones and (ii) penalise stations which exceed the cycle time. The change between testing shifts or swaps in an iteration and the heaviness of the punishment of overloads is managed by dynamically adapting parameters. Lapierre et al. use simple tabu rules for task reassignments and loading of stations.

Experience: In our experiments the algorithm suffered on three major weaknesses. First, the shift steadily empties stations without being able to pay much attention to overloads. Thus, the overload punishment parameter p rises very fast and reaches values even exceeding the double data type (≈1.8E+308) soon. So, p becomes equal to a heavy punishment factor M what it was not designed for. Second, Lapierre et al. state that empty stations must be closed and cannot be reopened. With the neighbourhoods and parameters suggested in Tabu-L it needs usually less than 100 shifts to have less open stations than the optimal solution, and so further calculations never lead to any improvements. Therefore, our implementation always keeps $m - 1$ stations open where m is the best upper bound found by Tabu-L so far (Scholl & Voß, 1996). Third, the tabu setting of tasks (and not task-station combinations as in Tabu-SV) for 25 ± 5 iterations is very restrictive. So in a large number of iterations even a shift is not allowed. Because Lapierre et al. report distinctly better results for the 26 instances of the graph Scholl in SCHOLL than we could find, one may assume that they have implemented several procedures different to their description.

2.8. Fleszar and Hindi (2003): MultiHoff

General idea: The well-known Hoffman Heuristic (Hoffmann, 1963) seeks at first for the best load of the first station as long as

none with zero-idle-time is found. Then with the remaining tasks the best load for the second station is calculated and fixed and so on. This conducted in forward and backward direction leads to exactly two solutions. Fleszar and Hindi propose a faster recursive implementation of the Hoffmann Heuristic and use a bidirectional search. Bidirectional means that at first $0,1,2,3,\ldots,m$ stations are loaded in the forwards (backwards) direction and the remaining tasks are assigned in the backward (forward) direction; i.e. approx. $2m$ solutions in total. Additionally they propose seven problem reduction techniques as add-on. Sternatz (2014) recently published a distinctly improved version of MultiHoff which is not tested here. His enhanced MultiHoff does not enumerates tasks by decreasing tasks times but by more "intelligent" priority rules until the first zero-idle-time loads are found.

Experience: The problem reduction techniques require much time for the implementation and are very prone to difficult to spot bugs. Appendix A clarifies and in some occasions corrects the description of SALBP-1 reduction techniques given by Fleszar and Hindi.

2.9. Bautista and Pereira (2009): Bounded-DP

General idea: Bautista and Pereira also build their Bounded Dynamic Programming on the Hoffmann Heuristic. Let us assume one knows not only one (as in the Hoffmann Heuristic) but a set of distinct partial solutions which load the first k stations, then (1) one selects at first the best b partial solutions of them according to their idle times in station k, (2) for each of these partial solutions one enumerates over the station loads for $k+1$ until either z zero-idle-time loads are found or the enumeration comes to an end, and (3) collects every found partial solution with $k+1$ stations in a pool.[1] From this pool the best b partial solutions are now selected to construct partial solutions with $k+2$ stations, and so forth. The procedure starts with $k=1$ and increases the number of stations until the first complete solution is found. Additionally, the lower bounds LB1 till LB3 are applied (see Appendix B) to reject poor partial solutions, the Hoffmann Heuristic is used as initial upper bound, duplicate partial solutions are eliminated on each stage k and the procedure repeated in the backward direction.

Experience: Instead of pooling the partial solutions in a list to eliminate duplicates by pairwise comparisons, we store them in a tree following the encoding rule from Nourie and Venta (1991) what allows quicker lookups.

2.10. t-Bounded-DP

Timed Bounded Dynamic Programming (t-Bounded-DP) builds on Bounded-DP and allows an approximation of the used CPU time.

Instead of measuring the quality of a partial solution (state) in step (1) of Bounded-DP by the idle time of the station under construction, we use the total idle time of the partial solution for t-Bounded-DP. This change makes also lower bounds LB1 to LB3 quasi redundant, because solutions refused by the weak LB1 till LB3 would be normally not considered for the next stage anyway.[2] To lower the CPU time, LB1 till LB8 as global stopping criterion when having reached an optimum are applied and the search for an initial solution is conducted with MultiHoff instead of the Hoffmann Heuristic.

In Table 2 the found results are compared with those reported by Bautista and Pereira. It shall be highlighted that the average and maximum time grows almost linearly to the given parameter b with gradient 1 in our implementation and not exponentially. One can exploit this relation to predict parameter settings which are near to a given CPU time limit. For each direction it is started with a trial for $b=z=10$ and the required time is stopped. With this information parameter b is estimated by linear approximation in a way that let the search in forward direction end after 50% of the given CPU time limit and the search in backward direction when reaching the time limit. To avoid imprecise predictions for extreme cases, a further trial with $b := 10b$ and $z=10$ is conducted if the estimated b is more than 100-fold higher than the tried one. If the estimated b is not larger than the already tried one, no further calculations are necessary. Figs. 2a and 2b show, respectively, the really used CPU times and the average b parameters for given time targets of 10 seconds on OTTO-100 and 180 seconds on OTTO-1000 when only considering instances which are not stopped before finishing the final search in backward direction. The relatively high concentration between zero and five seconds on OTTO-100 is present through instances of order strength 0.9. Those instances do usually not use up the entire size b of the pool of partial solutions with k stations from which partial solutions with $k+1$ stations are constructed. Obviously the approximation systematically underestimates the needed CPU time. Therefore the tests are given a limit of only 90% of the targeted CPU time, e.g. 162 seconds when targeting 180 seconds.

2.11. Blum (2008): Beam-ACO

General idea: In principle similar to the later published Bounded-DP, Blum takes a set of partial solution with the k first stations loaded, constructs out of each of them b_{ext} solutions with $k+1$ stations and collects them in a pool, and finally selects the best b_{best} solutions of this pool to construct partial solutions with $k+2$ stations. Which tasks shall be added to station $k+1$ is decided similarly as in ACO. Blum uses (i) task-station policy (i.e. pheromone trails τ_{jk} between tasks and stations), (ii) summed trail reading (i.e. $\sum_{k=1}^{\bar{k}}\tau_{jk}$ as decision criterion to assign task j to station $\bar{k}$), (iii) η_j as combined priority rules processing time and number of direct successors, (iv) switches between assigning tasks according to their preferences $\eta_j\sum_{k=1}^{\bar{k}}\tau_{jk}$ deterministically or with roulette wheel selection, and (v) applies LB1 to exclude non-promising partial solutions. For CPU times up to one minute $b_{ext}=b_{best}=10$ is used here, and for higher time limits $b_{ext}=b_{best}=20$ as suggested by Blum.

Experience: The ACO components of the procedure seem to deliver useful guidance but are not the main driver of success. For instance, a 360 seconds time limit with updating pheromones delivers 250 optimally solved instances on SCHOLL whereas without updating pheromones levels one detects 244. Furthermore, restarting Beam-ACO 12 times with 15 seconds CPU time on OTTO-100 produces a best solution which requires on average 0.06 stations less than executing Beam-ACO once with 180 seconds. This observation provides some evidence that also Beam-ACO suffers on convergence.

2.12. Scholl and Klein (1997, 1999): SALOME

General idea: SALOME is a station-oriented, depth-first, bidirectional branch-and-bound algorithm. Station-oriented means that branching is not done with respect to single task-station assignments but according to station loads as nodes. Thereby those station loads which do not induce raising the problem's lower bound are explored immediately and the others are the last developed branches of the current node. The lower bounds 1, 2, 3, 5, 6 and 7 are calculated for each sub problem and LB4 together with the heads and tails once in the root (see Appendix B). Additionally, SALOME includes many logical tests like tree dominance rule (Nourie & Venta, 1991) or extended Jackson's dominance rule to

[1] The parameters b and z are called window_size and max_transitions, respectively, by Bautista and Pereira (2009).

[2] In our experiments, the results and times reported in Table 2 do not depend on the usage of LB1 till LB3 at all.

Table 2
Results as detected in our slightly modified version and as reported by Fleszar and Hindi (2003) with number of found optima (#), average CPU time (Ø) and CPU time for the slowest instance (max) in seconds.

Parameter setting		Modified version (2.8 gigahertz)			Reported in Bautista and Pereira (2.4 gigahertz)		
b	*z*	#	Ø	max	#	Ø	max
10	5	236	0.06	1	227	0.09	1
50	5	255	0.3	2.75	254	0.4	4
100	10	257	0.9	9	260	1.5	12
250	10	266	2.2	24	264	3.6	26
500	10	267	4.4	48	265	9.8	64
750	10	268	6.4	75	267	30.2	182
1000	10	268	8.4	95	268	114	596
3000	20	269	51	572	Not reported	Not reported	Not reported

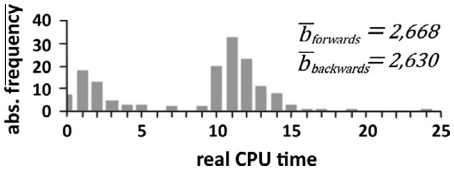


Fig. 2a. Real CPU time with 10 seconds on non-disrupted OTTO-100 instances for t-Bounded-DP.

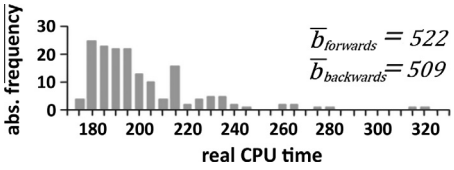


Fig. 2b. Real CPU time for 180 seconds on non-disrupted OTTO-1000 instances for t-Bounded-DP.

avoid branching of redundant or inferior partial solutions. Traditionally, SALBP-1 solution procedures search successively in forward and backward direction. SALOME constructs just one branch-and-bound tree and decides in each node by means of priority rules whether to look in the forward or backward direction. Despite SALOME is designed as exact procedure it can be applied as heuristic by limiting the CPU time. Finally, two recent improvements of the original SALOME design must mentioned which are not considered here. Sewell and Jacobson's branch, bound and remember algorithm (Morrison, Sewell, & Jacobson, 2013; Sewell & Jacobson, 2012) memorises a large list of previously solved sub problems to avoid redundant computations. Vilà and Pereira (2013) develop branches in order of increasing idles times, establish a new logical test based on the maximum flow problem, and incorporate a modified version of extended duration augmentation rule (Fleszar & Hindi, 2003).

Experience: The given CPU time is consumed most by the extended Jackson's dominance rule (24%) and the lower bounds (31%) when run for 10 seconds on OTTO-100. We also tested the original Pascal code from Scholl and Klein (1999) on the same computer as the new one. Thereby, the new object-oriented implementation was distinctly faster; e.g. on OTTO-100 with 180 seconds time limit the average deviation to the best-known lower bound could be improved from 0.91 to 0.41.

2.13. Random Search

The Random Search creates order encoded solutions in forward direction by randomly selecting not-assigned tasks which do not have any not-assigned predecessors and puts them at the end of the task sequence having found by now. Additionally the simple maximum load rule (Jackson, 1956) is applied, i.e. those tasks which still fit in the current station are preferred to those which demand the closing of the current station to open a new one.

Although the algorithm works with about 15,000 solutions per second on OTTO-100 quite fast, it must be noted that there is a dependency between the moment a task is added to the set of assignable tasks and the sequence position it is finally assigned to. Fig. 3 demonstrates that fact. In this example the best solution (task 6 after task 5) has the least probability. To reduce this effect, the task sequences are alternating generated in forward and backward direction.

2.14. Random Task Priority Search

Random Search assigns the next task from the list of available tasks randomly. As alternative one could apply a roulette wheel selection according to the task priorities raised to the power $\beta = 25$ ($\beta = 45$ for OTTO-1000 only).

Before starting the search, normalised priority values are computed. Let η_{rj} be the priority value for task j and rule r. For each rule r separately, the η_{rj} are normalised to values within the interval (Baybars, 1986; Scholl, 1999). So $\min_j\{\eta_{rj}\} = 1$ and $\max_j\{\eta_{rj}\} = 2$ $\forall r$. Thereby the rules processing time, number of successors, number of direct successors and positional weight with priorities t_j, $|S_j|$, $|S_j^*|$ and $t_j + t(S_j)$, respectively, are applied.

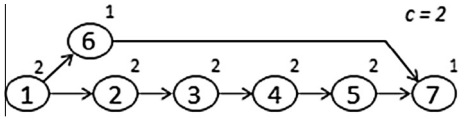
Every single assignment of an available task j to the next order position consists of two steps. At first one of the rules r is randomly selected, and at second a roulette wheel selection over η_{jr}^{β} is performed to decide on the next task j.

3. Lower bounds

If the lower bound (LB) of a problem equals the upper bound, i.e. best-known solution value, the upper bound is proven optimal and the calculation can be disrupted immediately. Thus, strong lower bounds are valuable to assess the quality of a solution and to reduce computational time distinctly.

3.1. Traditional lower bounds LB1 to LB7

For readers who are not fully familiar with the traditional lower bounds 1 till 7, we provide a brief summary in Appendix B based on the descriptions in Scholl (1999), Scholl and Klein (1997), and Sprecher (1999).



Probability that task 6 is placed after task j:

j	1	2	3	4	5
p_j	1/2	1/4	1/8	1/16	1/16

Fig. 3. Dependency in Random Search.

3.2. Tails and heads

A task's tail estimates the minimal time required for a workpiece to be fully assembled after this task has been completed. The tail is computed as lower bound of this task's successors adjusted by whether the task can join a station with any of its successors. Analogue, a task's head provides the minimum station time requirement of its predecessors.

Johnson (1988) introduces the traditional tails τ_{1j}, τ_{2j}, τ_{3j} and τ_{4j} derived, respectively, from LB1, LB2, LB3 and the one-machine scheduling problem.

A further tail τ_{5j} is suggested in Fleszar and Hindi (2003) based on the time requirement of the tasks which have to lie between two tasks with respect to the precedence graph: Let be $j \in P_{j'}$ and $\lambda_{jj'}$ the unrounded station requirement of the set $S_j \cap P_{j'}$ containing all tasks which have to lie between task j and j', then $\tau_{5j} = \max_{j'}\{\lambda_{jj'} + p_{j'} + \tau_{j'}\}$ with $p_{j'} = t_{j'}/c$. $\lambda_{jj'}$ equals the maximum of $\tau_1(S_j \cap P_{j'}), \tau_2(S_j \cap P_{j'})$ and $\tau_3(S_j \cap P_{j'})$. We also sharpen $\lambda_{jj'}$ with τ_6 as described next.

The new tail τ_{6j} applies LB6 to the successors of task j and explains the necessary adjustments to check whether task j itself fits in the tail's stations. Remember that there are three types of stations which are in the focus of LB6: d_1 stations (denoted as D_1) with the tasks from $J(0.5,1]$. d_2 stations (denoted as D_2) with the tasks from $J(\frac{1}{3},0.5]$ having not fitted in D_1. And d_3 stations (denoted as D_3) if there exist tasks from $J[q, 1-q]$ which do not fit in the stations from $J(0.5,1-q]$ and D_2, i.e. $d_3 = \max\{LB1(J[q,1-q]) - d_2 - |J(0.5,1-q]| | q \in J[0,\frac{1}{3}]\}$. So $LB6 = d_1 + d_2 + d_3$. If $d_3 = 0$, all stations are loaded to more than one-third when D_2 contains an odd number of tasks and to more than one-half when D_2 contains an even number of tasks. That means after the correction, which incorporates the lowest possible load for the tail's first station, one obtains a tail

$$\tau_{6j}(d_3 = 0) = \begin{cases} \lceil LB6(S_j)\rceil - \frac{1}{3}, & D_2 \text{ contains even number of tasks and } D_1 = \emptyset \\ \lceil LB6(S_j)\rceil - 0.5, & D_2 \text{ contains even number of tasks and } D_1 \neq \emptyset \\ \lceil LB6(S_j)\rceil - \frac{2}{3}, & \text{otherwise} \end{cases}.$$

A more interesting case is $d_3 > 0$ where there is at least one $q \in [0,\frac{1}{3}]$ in which the unrounded $d_3(q) = \sum_{j\in J[q,1-q]} p_j - d_2 - |J(0.5,1-q]| > 0$. This equation says that there are $\gamma_1(q) = \lceil d_3(q)\rceil + d_2 + |J(0.5,1-q]|$ stations (denoted as Γ_1) needed for the tasks $J[q,1-q]$ and, of course, also $\gamma_2 = |J(1-q,1]|$ separate stations (denoted as Γ_2) for the tasks from $J(1-q,1]$. That means in the worst case, the first station of Γ_1 has an idle time of $\gamma_1(q) - \sum_{j\in J[q,1-q]} p_j = \lceil d_3(q)\rceil - d_3(q)$ since (i) all other stations are filled completely and (ii) $J[0, q) = \emptyset$. About the stations belonging to Γ_2, one knows that they are loaded at least with the station requirement $\min\{p_j | j \in J(1-q,1]\}$ of the smallest task larger than $1-q$ or do not exist if $J(1-q,1] = \emptyset$. Now $d'_3 = \max\{d'_3(q) | q \in J[0,\frac{1}{3}]\}$ with $d'_3(q) = \lceil d_3(q)\rceil - \max\{d_3(q) - d_3(q), 1 - \min\{p_j | j \in J(1-q,1]\}\}$; where $\lceil d_3(q)\rceil - d_3(q)$ is the correction term for Γ_1, $1 - \min\{p_j | j \in J(1-q,1]\}$ the correction term for Γ_2, and the larger one of both is applied. It follows $\tau_{6j}(d_3 > 0) = d_1 + d_2 + d'_3$.

3.3. Lower bound 7a

LB7 takes the $d+1$ shortest of the $d \cdot m + 1$ tasks with the highest processing time and tests whether they fit in one station according to LB1, otherwise increases m by one. We strengthen this bound by trying to reject every station load not just with LB1 but also with the precedence graph. The next two paragraphs explain LB7a for the special case $d = 1$ at first.

Pseudocode 1. LB7a.

```
Function LB7a(m)
  For h = 1 to 15
    Get T as h'th smallest task tuple
    If T fulfils (i) Then Return m + 1
    If T does not fulfil (ii) and (iii) Then Return m
  EndFor
  Return m
```

Let be $\langle j_1, j_2, \ldots, j_m, j_{m+1}]$ the set of the $m+1$ largest tasks of a SALBP-1 instance ordered by increasing task times. Then (j_1, j_2) is the pair with the smallest task time sum, (j_1, j_3) with the second smallest, (j_1, j_4) or (j_2, j_3) with the third smallest, (j_1, j_4), (j_1, j_5), (j_2, j_3) or (j_2, j_4) with the fourth smallest, and so forth. In general, a task pair (j_{r_1}, j_{r_2}) can be the h'th smallest if $r_1 + r_2 \leqslant h + 2$. Thereby j_{r_i} is the task with the r_i highest task time. Exploiting this observation, it is not time consuming to compute the 15[3] smallest task pairs from $\langle j_1, j_2, \ldots, j_m, j_{m+1}]$.

A task pair $T = (j, j')$ with $j \in P_{j'}$ does not fit in one station if (i) $t_j + t_{j'} > c$, (ii) $L_j < E_{j'}$ or (iii) $t_j + t(S_j \cap P_{j'}) + t_{j'} > c$. Condition (ii) and (iii) reject only the pair (j, j') as possible load, condition (i) rejects (j, j') and every larger pair. Pseudocode 1 shows the behaviour of LB7a.

The general case with $d \geqslant 1$: Let be $\langle j_1, j_2, \ldots, j_{dm}, j_{dm+1}]$ the set of the $dm+1$ largest tasks ordered by increasing task times and $T = (j_{r_1}, j_{r_2}, \ldots, j_{r_d}, j_{r_{d+1}})$ a tuple with tasks from this set. T can be the h'th smallest tuple if $\sum_{i=1}^{d+1} r_i \leqslant h + \sum_{i=1}^{d+1} i - 1$. Using pseudocode 1 again, one increase m by one if (i) $\sum_{j\in T} t_j > c$, and rejects T as load if (ii) $L_j < E_{j'}$ for any task pair (j, j') with $j \in P_{j'}$ and $j, j' \in T$ or (iii) $\sum_{j\in T} t_j + t(\bigcup_{j,j'\in T} S_j \cap P_{j'}) > c$.

3.4. Lower bounds 8a till 8d

A new lower bound is proposed by Fleszar and Hindi (2003), again, based on the time requirement of the tasks which have to lie between two tasks to obey the precedence relations. The following is tested for all station pairs (k_1, k_2) with $1 \leqslant k_1 \leqslant k_2 \leqslant m$: Let B_{k_1,k_2} be the set of all tasks j which fulfil the inequalities $k_1 \leqslant E_j$, $L_j \leqslant k_2$, then all tasks in B_{k_1,k_2} must be allocated among the stations $k_1, \ldots, k_2$. For B_{k_1,k_2}, none of the lower bounds 1 till 3 is allowed to exceed $k_2 - k_1 + 1$ stations. Otherwise, the lower bound m can be increased by 1.

Lower Bound 8 separates numerous subproblems from the original graph and allows therefore an easy integration of every other bounding technique. We sharpen LB8 by (i) using LB6 instead of LB1, LB2 and LB3 to calculate the station requirement of B_{k_1,k_2} (→LB8a), (ii) additionally applying τ_{6j} to receive the heads and tails of task j (→LB8b), (iii) additionally calculating $LB7a$ of B_{k_1,k_2} (→LB8c) and (iv) additionally using τ_{5j} (→LB8d).

3.5. LBR: LB8c with reduction techniques

Fleszar and Hindi (2003) remark that reduction techniques have a significant impact on lower bounds. Their reduction techniques are initialised with a strong upper bound m and try to simplify the problem by assuming that there exists a feasible solution with m-1 stations. If (i) a lower bound calculated for the reduced problem is higher than $m-1$ or (ii) one task is never part of a feasible packing which does not exceed the total idle time, the assumption is revealed as false by contradiction.

[3] We suggest 15 as subjective trade-off between decreasing additional effectiveness of LB7a and exponentially increasing CPU time.

Table 3
Results for SCHOLL as tuple "number of found optima (average deviation to optima, maximum deviation to optima)" and for OTTO-100/1000 as "average deviation to best-known lower bound".

	SCHOLL			OTTO-100			OTTO-1000		
	1 second	15 seconds	180 seconds	1 seconds	15 seconds	180 seconds	30 seconds	180 seconds	900 seconds
GA-FD [a]	Does not find solutions for every instance			Does not find solutions for every instance					
GA-S [a]	122 (1.06, 6)	129 (0.76, 5)	136 (0.73, 5)	3.45	3.27	3.06			
DEA [a]	124 (1.38, 9)	136 (1.11, 8)	143 (0.95, 6)	4.15	3.13	2.86			
ACO	170 (0.45, 3)	190 (0.35, 2)	193 (0.31, 2)	1.41	1.07	0.91		Not tested	
Tabu-L	55 (2.91, 14)	69 (2.77, 14)	85 (2.58, 14)	6.73	6.49	6.31			
Tabu-SV	191 (0.37, 5)	200 (0.29,3)	197 [b] (0.31, 4)	1.54	0.52	0.46			
Simple-Tabu	237 (0.12, 1)	243 (0.1, 1)	244 [c] (0.09, 1)	0.31	0.27	0.21	10.16	4.88	4.19
MultiHoff	203 (0.27, 2)	Finishes in under 0.16 seconds		0.77	Finishes in under 0.19 seconds		11.32	Finishes in under 23.5 seconds	
MultiHoff with reduction	204 (0.26, 2)	Finishes in under 1 seconds		0.61	Finishes in under 0.42 seconds		11.12	10.64	10.63
t-Bounded-DP	251 (0.07, 1)	266 (0.01, 1)	268 (0.00, 1)	0.34	0.29	0.23	Time limit too low	7.73	7.08
Beam-ACO	231 (0.14, 1)	243 (0.10, 1)	248 (0.08, 1)	0.53	0.39	0.38	9.26	8.67	8.19
SALOME	243 (0.1, 1)	256 (0.05, 1)	259 (0.04, 1)	0.5	0.44	0.38	5.59	5.54	5.5
Random Search	103 (1.11, 5)	116 (0.94, 4)	124 (0.86, 4)	2.83	2.49	2.24	31.46	30.92	30.47
Random Task Priority Search	192 (0.3, 2)	199 (0.26, 2)	205 (0.24, 2)	1.08	0.88	0.75	18.71	18.3	17.86

[a] Randomly-generated initial population excluded.
[b] Cycle entrance depends on parameter setting and within the given CPU time.
[c] Simple-Tabu could only solve 4 out of 21 instances of the precedence graph Scholl in SCHOLL.

We exploit this mechanism by assuming that there exists a feasible solution with an amount of stations equal to the best-known lower bound and applying the reduction techniques. If a contradiction occurs, the lower bound is increased by 1 and reduction revoked (destructive improvement); otherwise the lower bound is accepted. LB6 is tested after every single reduction and the time-consuming LB8c only after no further reduction can be found for the present lower bound.

4. Computational results

4.1. Computational environment

The algorithms were coded in VB.NET x64 release and ran on a 2.8 gigahertz processor with 4 gigabytes RAM. Direct and indirect predecessors and successors were saved in a list as well as a 0–1 matrix and created before starting the stopwatch.

4.2. Heuristics

Table 3 shows the computational results on all 3 datasets. The results for SCHOLL are compared with the optimal solution values and for OTTO-100/1000 with the best-known lower bounds.

Random Search distinctly outstripped GA-FD and Tabu-L, as well as yielded results in the proximity of DEA and GA-S. Random Task Priority Search was slightly superior to ACO and reached MultiHoff on SCHOLL and OTTO-100 in the long run. MultiHoff always finished before reaching the lowest given time limits and was distinctly boosted on OTTO-100/1000 when using reduction techniques. The changes on Tabu-SV towards Simple-Tabu could break cycling, double the number of iterations, and thus improve results. For larger CPU times Simple-Tabu and t-Bounded-DP together clearly dominated all other heuristics (except of SALOME for instance 297 from OTTO-100).

OTTO-100/1000 can be grouped by three properties. The graph structure contains chains of tasks which have only one successor each (CH), bottleneck tasks with many direct predecessors and successors (BN), or no willingly constructed features (MIX). The task times are normally distributed with a peak at $0.1c$ (BOT), with a peak at $0.5c$ (MID), or bimodal with peaks at $0.1c$ and $0.5c$ (BI). The order strength is 0.2, 0.6 or 0.9. Table 4 shows the results split into all existing combinations of these properties for Simple-Tabu and t-Bounded-DP on OTTO-100/1000 with a time limit of 3600 seconds. Simple-Tabu delivered strong results for MID instances, especially those with low order strength; whereas t-Bounded-DP worked better in all other cases. The average deviation to the best-known lower bound is 0.2/3.18 and 0.21/6.19 for Simple-Tabu and t-Bounded-DP, respectively, for 3600 seconds CPU time on OTTO-100/1000.

4.3. Lower bounds

Table 5 contains the results for LB1 till LB8. "Found LBMax" counts the number of cases where no other lower bound produced a better result, "Unique LBMax" where no other lower bound computed an equal or better result, and "Found optima" where the lower bound reached the optimal solution value. The column LBMax gives the results when combining LB1 till LB8 and avoiding redundant calculations. LB5, LB7 and LB8 were initialised with LB1. It shall be highlighted that LB1 found always the currently best-known lower bound for any combination of BOT or BI with order strength 0.2 or 0.6 in OTTO-100/1000. All "unique LBMax" of LB6 had the task time distribution MID and of LB8 the order strength 0.9.

Table 6 summarises the results for the improved versions. Thereby LB7a, the improvements in LB8a, LB8b and LB8c, as well as the application of reduction techniques paid off. LB8d's calculation of the sets $S_j \cap P_{j'}$ was time consuming but did not breed success. All improvements for OTTO-100/1000 were for instances with task time distribution MID.

Combining LB8c with reduction techniques lead to strong improvements as displayed in the column LBR. Table 7 compares LBMax and LBR with the best-known upper bound[4] grouped by problem characteristics.

5. Discussion

We compared existing heuristics and lower bounds for SALBP-1 on a traditional and a lately-published benchmark dataset, as well as improved the best procedures. To our knowledge, it is the only study of this kind.

[4] Based on the results from our experiments and those reported in Morrison et al. (2013).

Table 4
Results for Simple-Tabu and t-Bounded-DP with 3600 seconds running time grouped by the instance properties and reported as "average deviation to best-known lower bound".

	MIX			CH		BN	
	0.2	0.6	0.9	0.2	0.6	0.2	0.6
OTTO-100							
BOT							
Simple-Tabu	0	0	0.08	0	0	0	0
t-Bounded-DP	0	0	0	0	0	0	0
BI							
Simple-Tabu	0.04	0	0	0.08	0.08	0	0.04
t-Bounded-DP	0	0	0	0	0	0	0.04
MID							
Simple-Tabu	0.12	0.8	1	0.24	1	0.12	0.6
t-Bounded-DP	0.24	0.76	1	0.4	0.92	0.4	0.6
OTTO-1000							
BOT							
Simple-Tabu	0	0.08	0.56	0.04	0.2	0	0.04
t-Bounded-DP	0	0	0	0	0	0	0
BI							
Simple-Tabu	0.12	0.24	1.68	0.24	0.64	0.12	0.4
t-Bounded-DP	0	0	0.24	0	0	0	0
MID							
Simple-Tabu	1.64	5.36	41.84	2.56	7.76	1.08	2.28
t-Bounded-DP	17.16	18.24	19.88	18.96	18.16	18.08	19.32

Table 5
Results for LB1 till LB8 in "number of instances from library".

	LB1	LB2	LB3	LB4	LB5	LB6	LB7	LB8	LBMax
SCHOLL									
Found LB-Max	185	15	12	248	242	217	204	252	–
Unique LB-Max	0	0	0	0	0	8	3	3	–
Found optima	123	11	11	185	179	152	141	188	201
OTTO 100									
Found LB-Max	355	11	13	363	358	502	0	408	–
Unique LB-Max	0	0	0	0	0	110	0	15	–
Average time in ms	0	0	0	0.32	0.29	0.01	0.02	0.79	0.84
OTTO 1000									
Found LB-Max	353	8	7	353	353	521	360	374	–
Unique LB-Max	0	0	0	0	0	149	0	4	–
Average time in ms	0	0.01	0.01	23.79	23.02	0.12	0.87	179.7	181.2

Table 6
Results for improved lower bounds in "number of instances from library".

	LB7a	LB8a	LB8b	LB8c	LB8d	LBR
SCHOLL						
Improvements to original LB	3	14	14	17	17	–
Improvements to LBMax	0	0	0	0	0	37
Found optima	143	199	199	201	201	234
OTTO 100						
Improvements to original LB	29	135	137	139	139	–
Improvements to LBMax	6	21	26	29	29	97
Average/ maximum time in ms	0.15/0.48	1.4/8	1.7/14	5.5/64	14.4/76	178/866
OTTO 1000						
Improvements to original LB	25	156	156	158	158	–
Improvements to LBMax	4	12	16	20	20	75
Average/ maximum time in sec	0.02/0.05	0.47/5.67	0.49/8.48	2.1/33.9	10.1/74	27.9/643

The computational tests identified Timed Bounded Dynamic Programming and Simple Tabu Search as most effective heuristics for SALBP-1; each for different problem characteristics.

Timed Bounded Dynamic Programming changes some details of Bounded Dynamic Programming (Bautista & Pereira, 2009) and seeks parameter settings which let it finish after reaching a targeted CPU time. The procedure distinctly extends the search space of the original Hoffmann Heuristic (Hoffmann, 1963) and is the first heuristic which can solve all 269 instances of Scholl's library (Scholl, 1993) in just a few minutes. The algorithm gets into enormous troubles when it comes to instances with many large tasks (here $t_j \sim \mathcal{N}(0.5c, 0.0025c^2)$ approx.). The poor results are most likely caused by the greediness of the original Hoffmann Heuristic which wants to load each station as full as possible without weighing up the consequences for the next stations. But these problem characteristics are quite uncommon in practical applications (Otto et al., 2013). Furthermore when confronted with problems of order strength 0.9 from 100-task instances, Timed Bounded Dynamic Programming is often not able to find a larger amount of feasible loads for many stations and therefore finishes long before reaching the time limit.

Simple Tabu Search simplifies and randomises Scholl and Voß' tabu search (Scholl & Voß, 1996), and thus improves its performance. Among all tested heuristics it yielded the best average

Table 7
Results for LBMax and LBR grouped by the instance properties and reported as "average deviation to best-known upper bound [number of instances with lower bound = best-known upper bound]".

	MIX			CH		BN	
	0.2	0.6	0.9	0.2	0.6	0.2	0.6
OTTO-100							
BOT							
LBMax	0 [25]	0 [25]	0.28 [18]	0 [25]	0 [25]	0 [25]	0 [25]
LBR	0 [25]	0 [25]	0.08 [23]	0 [25]	0 [25]	0 [25]	0 [25]
BI							
LBMax	0 [25]	0.04 [24]	0.56 [11]	0 [25]	0 [25]	0 [25]	0 [25]
LBR	0 [25]	0.04 [24]	0.32 [17]	0 [25]	0 [25]	0 [25]	0 [25]
MID							
LBMax	0.44 [16]	1.56 [2]	2.92 [0]	0.36 [16]	1.76 [2]	0.28 [18]	0.04 [24]
LBR	0.28 [18]	0.72 [11]	1 [4]	0.32 [17]	0.96 [7]	0.12 [22]	0.04 [24]
OTTO-1000							
BOT							
LBMax	0 [25]	0 [25]	0 [25]	0 [25]	0 [25]	0 [25]	0 [25]
LBR	0 [25]	0 [25]	0 [25]	0 [25]	0 [25]	0 [25]	0 [25]
BI							
LBMax	0 [25]	0 [25]	0.24 [19]	0 [25]	0 [25]	0 [25]	0 [25]
LBR	0 [25]	0 [25]	0.24 [19]	0 [25]	0 [25]	0 [25]	0 [25]
MID							
LBMax	2.12 [2]	6.4 [0]	27.96 [0]	2.72 [1]	8.84 [0]	1.4 [8]	2.92 [2]
LBR	2 [2]	5.6 [0]	19.76 [0]	2.72 [1]	8 [0]	1.36 [8]	2.44 [3]

results for OTTO-100 and OTTO-1000. Higher order strength (0.6 and especially 0.9) sapped its solution quality. More precedence restrictions reduce the amount of possible moves in each iteration and so the local search algorithm cannot explore the whole solution space properly.

Our experiments give some evidence that establishing effective learning through metaheuristics may not be possible for SALBP-1. The only metaheuristics which could outperform the quite dull Random Task Priority Search were Simple Tabu Search (only with a short-term memory) and Beam-ACO (with enumeration as main driver of success). We speculate that learning fails because SALBP-1 solutions are sensitive to minor modifications. Slightly changing the assignment of a single task can easily mean (i) that direct successors of the task have to be shifted backwards or direct predecessors forwards to obey the precedence relations and (ii) that the task's new station is overloaded and whole other tasks (and not just portions of them) have to be shifted backwards or forwards. Both together often trigger self-enforcing domino-effects resulting in new stations which have to be opened. Thus, merging structural characteristics from several attractive solutions is rather unlikely to breed efficient new solutions for SALPB-1 and, perhaps, also for other constrained partitioning problems.

Pseudocode 2. Packing enumeration.

```
Main(j) {
AddLargerTask({j})

AddLargerTask(A) {
AvailabeTasks contains all tasks which can be added
  to A and have a larger index than the largest index
  in A
For each j in AvailableTasks {AddLargerTask(j ∪ A)}

AddSmallerTask(A)
AddSmallerTask(A) {
AvailabeTasks contains all tasks which can be added
  to A and have a smaller index than the smallest
  index in A
For each j in AvailableTasks {AddSmallerTask(j ∪ A)}
```

Simple Tabu Search as metaheuristic offers the advantages that it can be often used without many changes for general assembly line balancing problems (GALBPs) and that its CPU time can be scaled arbitrarily. It was shown that Timed Bounded Dynamic Programming does hardly rely on its lower bounds for SALBP-1. So if (i) one uses another fitness function than the total idle time to evaluate partial solutions and (ii) selects another stopping criterion than the number of found zero-idle-time loads when enumerating partial solutions with $k+1$ station from a given partial solution with k stations, it should be possible to adapt this procedure to many GALBPs (Bautista & Pereira, 2011). Furthermore, it was shown that its time requirement is with some margin of error quite well adjustable.

Among the lower bounds from the literature, 6 and 8 yielded the best results. The best-known lower bounds could be distinctly sharpened by (i) using LB6 to calculate heads and tails, (ii) strengthening LB7 with the precedence graph, (iii) integrating LB6 and LB7a in LB8 and especially (iv) applying reduction techniques.

Finally this study indicates the importance of benchmark datasets with well grouped problem characteristics. They can lead to insight into the strengths and weaknesses of an algorithm and proofed in this case that the success of SALBP-1 algorithms can distinctly depend on the problem properties.

Acknowledgements

I am grateful to Armin Scholl for suggesting the research topic and Nils Boysen for providing the code used in Boysen and Fliedner (2008). The paper also benefited from many detailed remarks of an anonymous referee.

Appendix A. Remarks on reduction techniques

The appendix shall clarify and in some occasions correct the description of SALBP-1 reduction techniques given by Fleszar and Hindi (2003). The following bullet points can be only understood together with their original paper.

- Fleszar and Hindi just mention that enumerating for all $\mathcal{A}_j$ works quite similar to their Hoffmann code. In it, the procedure OnePackingSearch relies on the fact that L as the set of available or already assigned tasks just expands with each new task added to the load. But adding tasks to a packing may make some tasks unavailable again. Therefore we apply a slightly different approach. In order to avoid several permutations of one packing, the tasks in the packing are ordered increasingly to their indexes (Johnson, 1988). The algorithm is shown in pseudocode 2. AddLargerTask ($\mathcal{A}$) extends the packing to the "right" (line 6) and hands over each found (not necessary maximal) packing $\mathcal{A}$ to AddSmallerTask in order to expand it to the "left" (line 7).
- AvailableTasks should be enumerated in the increasing order of the difference between their task indexes and j in the last lines of AddLargerTask($\mathcal{A}$) and AddSmallerTask($\mathcal{A}$). Therewith one avoids many packings like $\{2, j\}$ or $\{j, n-2\}$ which can be barely extended nor they are likely to have a low idle time. Not applying this strategy can easily increase the CPU time by factor 10.
- In order to eliminate non-conjoinable tasks quickly, one should seek for two maximal packings for each task at first. This can be done by identifying a packing when solemnly looking in the "right" direction, i.e. evoking AddLargerTask({j}) without line 5, and in the "left" direction, i.e. evoking AddSmaller Task({j}).
- Any maximal packing $\mathcal{A}_j$ is also a maximal packing for any other task $i \in \mathcal{A}_j$ and can update the conjoinable tasks for i.
- Let be C a set of tasks which shall be conjoined. Any maximal or zero-idle-time packing containing some or all tasks from C must be discarded. The same is true for those packings which would have the new conjoined task as successor and as predecessor.
- Fleszar and Hindi state that any packing which contains a successor and a predecessor of a task but not the task itself is invalid. To incorporate this rule efficiently, one should only allow the adding of those tasks $j \in S^*(\mathcal{A})$ to the packing $\mathcal{A}$ which do not have a predecessor which is also in $S^*(\mathcal{A})$ or those tasks $j \in P^*(\mathcal{A})$ which do not have a successor which is also in $P^*(\mathcal{A})$.
- Fleszar and Hindi call a packing $\mathcal{A}$ maximal if no task from $F'(\mathcal{A})$ can be added to $\mathcal{A}$ without exceeding the cycle time. They describe $F'(\mathcal{A})$ as the set of (i) all direct predecessors and successors of the packing joined with (ii) all tasks which are not related to the packing and do not have any relations to tasks the packing itself does not have. Formally they define $F'(\mathcal{A}) = \{j \in F(\mathcal{A}) | P_j \subseteq P(\mathcal{A}) \cup \mathcal{A} \text{ and } S_j \subseteq S(\mathcal{A}) \cup \mathcal{A}\}$, where $F(\mathcal{A})$ is the set of tasks which are either directly related or not related to $\mathcal{A}$. The more restrictive formula – which also excludes some direct successors or predecessors of the packing – should be preferred. Instance 248 of Otto et al.'s 20-task problem (Otto et al., 2013) can be seen as proof since their verbal description would allow joining tasks 4 and 8 and so increasing LB4 to a value larger than in the optimum.
- Regularly there is no packing $\mathcal{A}_j$ for task j with $c - t(\mathcal{A}_j) \leqslant I_{total}$. In these cases the reduction can be broken immediately and the upper bound increased by one.
- After Conjoining tasks or adding precedence relations, direct precedence relations of not involved tasks may be explained by new indirect precedence relations and must be eliminated therefore.

Appendix B. Traditional lower bounds

LB1: Station borders and precedence relations neglected; i.e. $LB1 = \left\lceil \sum_j t_j / c \right\rceil$.

LB2: All tasks from $J(0.5,1]$ require separate stations and 2 tasks each from $J[0.5,0.5]$ can join a station; i.e. $LB2 = |J(0.5,1]| + \lceil 0.5|J[0.5,0.5]| \rceil$.

LB3: LB3 uses the subsets $J\left(\frac{2}{3},1\right], J\left[\frac{2}{3},\frac{2}{3}\right], J\left(\frac{1}{3},\frac{2}{3}\right)$ and $J\left[\frac{1}{3},\frac{1}{3}\right]$. All tasks from $J\left(\frac{2}{3},1\right]$ can never be assigned to a station together with any task from the other three subsets. Two tasks from $J\left(\frac{1}{3},\frac{2}{3}\right)$ might share one station. A task from $J\left[\frac{1}{3},\frac{1}{3}\right]$ might be joined with a task from $J\left(\frac{1}{3},\frac{2}{3}\right)$, from $J\left[\frac{2}{3},\frac{2}{3}\right]$ or two further tasks from $J\left[\frac{1}{3},\frac{1}{3}\right]$. These station loads can be characterised by giving tasks from $J\left(\frac{2}{3},1\right], J\left[\frac{2}{3},\frac{2}{3}\right], J\left(\frac{1}{3},\frac{2}{3}\right)$ and $J\left[\frac{1}{3},\frac{1}{3}\right]$ a weight of 1, $\frac{2}{3}, \frac{1}{2}$ and $\frac{1}{3}$, respectively. $LB6 = \lceil |J\left(\frac{2}{3},1\right]| + \frac{2}{3}|J\left[\frac{2}{3},\frac{2}{3}\right]| + \frac{1}{2}|J\left(\frac{1}{3},\frac{2}{3}\right)| + \frac{1}{3}|J\left[\frac{1}{3},\frac{1}{3}\right]| \rceil$.

LB4: In the single-machine scheduling problem, orders $j = 1 \ldots n$ can only be produced successively on one machine. Each order requires a time t_j on the machine and a time τ_j for successive processes (tail). The makespan is the time between the first order comes into the machine and the tail for the last order is completed. To minimise the makespan, one starts with order j_1 with the longest tail (started at time 0 and finished at time $t_{j_1} + \tau_{j_1}$), followed by order j_2 with the second longest tail (started at time t_{j_1} and finished a time $t_{j_1} + t_{j_2} + \tau_{j_2}$), and so on. The makespan is the maximum of all end times, i.e. $\max\{t_{j_1} + \tau_{j_1}, t_{j_1} + t_{j_2} + \tau_{j_2}, \ldots, t_{j_1} + t_{j_2} + \cdots + t_{j_n} + \tau_{j_n}\}$.

In SALBP-1 the orders j are called tasks and the tails are formed by j's successors. The tail τ gives the number of stations (not necessarily integer) which must follow task j at least. $\tau_{1j} = t(S_j)/c$ is the unrounded LB1. $\tau_{2j} = LB2(S_j) - 0.5$ and $\tau_{3j} = LB2(S_j) - 1/3$ are derived, respectively, from LB2 and LB3 with corrections coming into place to take into account that there might be still one-half and one-third of the first station required by S_j free (see Fig. 4). $\tau_{4j} = \max\{p_{j_1} + \tau_{j_1}, p_{j_1} + p_{j_2} + \tau_{j_2}, \ldots, p_{j_1} + p_{j_2} + \cdots + p_{j_{|S_j|}} + \tau_{j_{|S_j|}}\}$ calculates the unrounded makespan for the subgraph of the tasks from S_j; i.e. $\tau_{4j} = LB4(S_j)$.

$\tau_j = \max_i\{\tau_{ij}\}$ for all used tail rules i. One can round up τ_j further to the nearest integer when $\tau_j + p_j > \lceil \tau_j \rceil$, i.e. the task j does definitely not fit in the tail's first station. Having found the tails of all tasks, one calculated the makespan of the entire graph with $LB4 = \max\{p_{j_1} + \tau_{j_1}, p_{j_1} + p_{j_2} + \tau_{j_2}, \ldots, p_{j_1} + p_{j_2} + \cdots + p_{j_n} + \tau_{j_n}\}$.

A task's head h_j gives the number of stations which need to come before the task in the final solution and equals the tail of the reverse SALBP-1 problem.

LB5: The earliest possible station for a task is not allowed to be higher than the latest possible station; i.e. $E(j) \leqslant L(j)$. Thereby, $E(j) = \lceil h_j + p_j \rceil$ and $L(j) = m + 1 - \lceil \tau_j + p_j \rceil$.

LB6: Assign the tasks $j \in J\left(\frac{1}{2},1\right]$ to single stations and order these by decreasing workload to obtain the station sequence $D_1 = [s_1, s_2, \ldots, s_{d_1}\rangle$. Arrange the tasks in $J\left(\frac{1}{3},\frac{1}{2}\right]$ by increasing task times to receive the ordered set $\langle j_1, j_2, \ldots]$. Afterwards, assign each task of $\langle j_1, j_2, \ldots]$ to the earliest possible station from $[s_1, s_2, \ldots, s_{d_1}\rangle$ starting with j_1.

If there is still a positive number of z tasks from $J\left(\frac{1}{3},\frac{1}{2}\right]$ which did not fit in any $s_1, s_2, \ldots, s_{d_1}$ one can treat them like in LB3. That means $d_2 = z/2$ additional stations $D_2 = \{s_{d_1+1}, \ldots, s_{d_1+d_2}\}$ need to be opened.

The remaining tasks from $J\left[0,\frac{1}{3}\right]$ can be assigned to the stations in D_1, in D_2 or – if necessary – to new stations. The stations $s_1, s_2, \ldots, s_{d_1}$ are at least loaded with the task times from $J(0.5,1]$. For the stations $s_{d_1+1}, \ldots, s_{d_1+d_2}$, it is only possible to say that there load exceeds $\frac{1}{3}$. LB6 exploits that

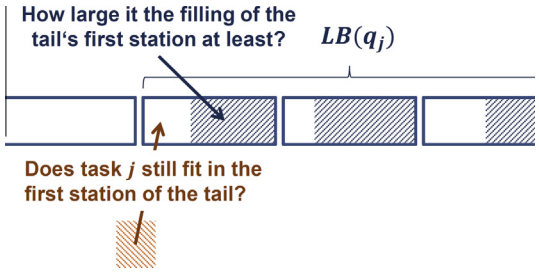


Fig. 4. Main idea of the correction problem.

the tasks $J\left[q,\frac{1}{3}\right]$ do not fit into those stations k from $s_1, s_2, \ldots, s_{d_1}$ with $1-p(k) < q$ but only in the $|J(0.5, 1-q]|$ stations from D_1 and the stations from D_2. I.e. $J[q, 1-q]$ must fit in the $|J(0.5, 1-q]| + |D_2|$ existing stations or new stations need to be opened. Therefore, $d_3\ (q) = t(J[q, 1-q])/c - |J(0.5, 1-q]| - d_2$ and $d_3 = \lceil \max_q\{0,\ d_3\ (q)\}\rceil$. $t(J[q, 1-q])/c$ is the unrounded LB1 of $J[q, 1-q]$.
It follows $LB6 = d_1 + d_2 + d_3$.

LB7: Let be r_i the task with the i highest task time and m the largest known lower bound. Among the $m+1$ most time consuming tasks at least two must be in one station (pigeonhole principle). This condition can be at best fulfilled by r_m and r_{m+1}; i.e. the inequality $t_{r_m} + t_{r_{m+1}} \leqslant c$ must hold. Otherwise, the lower bound m is not reachable and can be increased. This idea is generalised to the $dm+1$ most time consuming tasks. At least once, $d+1$ of them must join one station. This condition can be at best fulfilled by r_{dm+1-d}, $r_{dm+2-d}, \ldots, r_{dm+1}$; i.e. $\sum_{i=0}^{d} t(r_{dm+1-i}) \leqslant c$ must hold for all $d = 1, \ldots, \left\lfloor \frac{n-1}{m} \right\rfloor$ if m is the LB7.

Appendix C. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.ejor.2014.06.023.

References

Bautista, J., & Pereira, J. (2002). Ant algorithms for assembly line balancing. In *Ant algorithms, third international workshop, ANTS* (pp. 65–75). Brussels: Springer.

Bautista, J., & Pereira, J. (2007). Ant algorithms for a time and space constrained assembly line balancing problem. *European Journal of Operational Research, 177*(3), 2016–2032.

Bautista, J., & Pereira, J. (2009). A dynamic programming based heuristic for the assembly line balancing problem. *European Journal of Operational Research, 194*(3), 787–794.

Bautista, J., & Pereira, J. (2011). Procedures for the time and space constrained assembly line balancing problem. *European Journal of Operational Research, 212*(3), 473–481.

Baybars, I. (1986). A survey of exact algorithms for the simple assembly line balancing problem. *Management Science, 32*(8), 909–932.

Blum, C. (2008). Beam-ACO for simple assembly line balancing. *INFORMS Journal on Computing, 20*(4), 618–627.

Boysen, N., & Fliedner, M. (2008). A versatile algorithm for assembly line balancing. *European Journal for Operations Research, 184*(1), 39–56.

Falkenauer, E., & Delchambre, A. (1992). A genetic algorithm for bin packing and line balancing. In *Proceedings of the 1992 IEEE international conference on robotics and automation* (pp. 1186–1192). Nice.

Fleszar, K., & Hindi, K. S. (2003). An enumerative heuristic and reduction methods for the assembly line balancing problem. *European Journal of Operations Research, 145*(3), 606–620.

Gonçalves, J. F., & De Almeida, J. R. (2002). A hybrid genetic algorithm for assembly line balancing. *Journal of Heuristics, 8*(6), 629–642.

Hoffmann, T. R. (1963). Assembly line balancing with a precedence matrix. *Management Science, 9*(4), 551–562.

Jackson, J. R. (1956). A computing procedure for a line balancing problem. *Management Science, 2*(3), 261–271.

Johnson, R. V. (1988). Optimally balancing large assembly lines with "Fable". *Management Science, 34*(2), 240–253.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), *Complexity of computer computations* (pp. 85–103). New York: Plenum Press.

Lapierre, S. D., Ruiz, A., & Soriano, P. (2006). Balancing assembly lines with tabu search. *European Journal of Operational Research, 168*(3), 826–837.

Morrison, D. R., Sewell, E. C., & Jacobson, S. H. (2013). An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset. *European Journal of Operational Research, 236*(2), 403–409.

Nearchou, A. C. (2005). A differential evolution algorithm for simple assembly line balancing. In*16th International federation of automatic control (IFAC) World Congress*. Prague.

Nourie, F. J., & Venta, E. R. (1991). Finding optimal line balances with OptPack. *Operations Research Letters, 10*, 165–171.

Otto, A., Otto, C., & Scholl, A. (2013). Systematic data generation and test design for solution algorithms on the example of SALBPGen for assembly line balancing. *European Journal of Operational Research, 228*(1), 33–45.

Otto, A., Otto, C., & Scholl, A. (2014). How to design and analyze priority rules: Example of simple assembly line balancing. *Computers & Industrial Engineering, 69*, 43–52.

Ponnambalam, S. G., Aravindan, P., & Mogileeswar Naidu, G. (1999). A comparative evaluation of assembly line balancing heuristics. *International Journal of Advanced Manufacturing Technology, 15*(8), 577–586.

Sabuncuoglu, I., Erel, E., & Tanyer, M. (2000). Assembly line balancing using genetic algorithms. *Journal of Intelligent Manufacturing, 11*(3), 295–310.

Scholl, A. (1993). Data of assembly line balancing problems. *Schriften zur Quantitativen Betriebswirtschaftslehre 16/93, TU Darmstadt*.

Scholl, A. (1999). *Balancing and sequencing of assembly lines*. Heidelberg: Physica-Verlag.

Scholl, A., & Klein, R. (1997). SALOME: A bidirectional branch-and-bound procedure for assembly line balancing. *INFORMS Journal on Computing, 9*(4), 319–334.

Scholl, A., & Klein, R. (1999). Balancing assembly lines effectively – A computational comparison. *European Journal of Operational Research, 114*(1), 50–58.

Scholl, A., & Voß, S. (1996). Simple assembly line balancing – Heuristic approaches. *Journal of Heuristics, 2*(3), 217–244.

Sewell, E. C., & Jacobson, S. H. (2012). A branch, bound, and remember algorithm for the simple assembly line balancing problem. *INFORMS Journal on Computing, 24*(3), 433–442.

Sprecher, A. (1999). A competitive branch-and-bound algorithm for the simple assembly line balancing problem. *International Journal of Production Research, 37*(8), 1787–1816.

Sternatz, J. (2014). Enhanced multi-Hoffmann heuristic for efficiently solving real-world assembly line balancing problems in automotive industry. *European Journal of Operational Research, 235*(3), 740–754.

Vilà, M., & Pereira, J. (2013). An enumeration procedure for the assembly line balancing problem based on branching by non-decreasing idle time. *European Journal of Operational Research, 229*(1), 106–113.

Zhang, Z., Cheng, W., Tang, L., & Zhong, B. (2007). Ant algorithm with summation rules for assembly line balancing problem. In *14th International conference on management science & engineering* (pp. 369–374). Harbin (China).